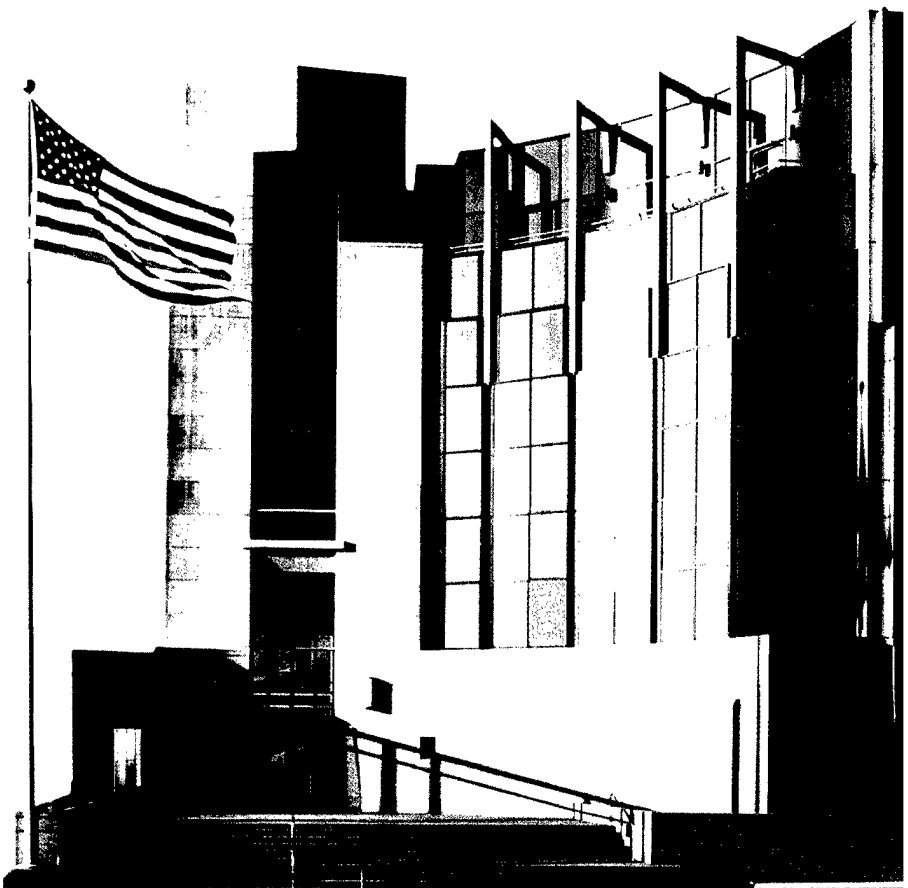
Carnegie Mellon
Software Engineering Institute

A Structured Approach to Classifying Security Vulnerabilities

Robert C. Seacord
Allen D. Householder

January 2005

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited



TECHNICAL NOTE
CMU/SEI-2005-TN-003

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

A Structured Approach to Classifying Security Vulnerabilities

CMU/SEI-2005-TN-003

Robert C. Seacord
Allen Householder

January 2005

Survivable Systems

Table of Contents

List of Figures

List of Tables

Abstract

Understanding vulnerabilities is critical to understanding the threats they represent. Vulnerabilities classification enables collection of frequency data; trend analysis of vulnerabilities; correlation with incidents, exploits, and artifacts; and evaluation of the effectiveness of countermeasures. Existing classification schemes are based on vulnerability reports and not on an engineering analysis of the problem domain. In this report a classification scheme that uses attribute-value pairs to provide a multidimensional view of vulnerabilities is proposed. Attributes and values are selected based on engineering distinctions that allow vulnerabilities to be exploited by a given technique or determine which countermeasures are effective. Successful classification of vulnerabilities should lead to greater automation in analyzing code vulnerabilities and supporting effective communication between geographically remote vulnerability handling teams and vendors.

1 Introduction

Historically, vulnerabilities have been classified into broad categories such as buffer overflows, format string vulnerabilities, and integer type range errors (including integer overflows). These broad categories have two major failings, however. First, it is not always possible to assign a vulnerability to a single category. Second, the distinctions are too general to be useful in any detailed engineering analysis.

For example, the following function:

```
bool func(char *s1, int len1,
          char *s2, int len2) {
  char buf[128];

  if (1 + len1 + len2 > 128) return false;

  if (buf) {
    strncpy(buf, s1, len1);
    strncat(buf, s2, len2);
  }
  return true;
}
```

contains a vulnerability in that `len1` or `len2` could be a negative number, allowing the length check to be bypassed but still causing a buffer overflow in the `strncpy()` or `strncat()` functions. Is this an integer range value vulnerability because the integer range check was bypassed, or is this simply a buffer overflow? Either categorization would be a disservice to understanding the issues.

Understanding vulnerabilities is critical to understanding the threats they represent. Classification of vulnerabilities allows collection of frequency data and trend analysis of vulnerabilities but has not been regularly or consistently applied. Better and more comprehensive classification of vulnerabilities can lead to better correlation with incidents, exploits, and artifacts and can be used to determine the effectiveness of countermeasures. Understanding the characteristics of vulnerabilities and exploits is also essential to the development of a predictive model that can predict threats with a high correlation and significance.

1.1 Concepts

Before we can discuss a classification scheme it is important that we have a sufficiently precise definition of what it is that we are classifying. There have already been efforts to formally define concepts such as vulnerability [Fithen 04] that will not be repeated here. For our purposes, we define four key terms using concise and (hopefully) precise English:

> *A **security flaw** is a defect in a software application or component that, when combined with the necessary conditions, can lead to a software vulnerability.*
>
> *A **vulnerability** is a set of conditions that allows violation of an explicit or implicit security policy.*
>
> *An **exploit** is a piece of software or a technique that takes advantage of a security vulnerability to violate an explicit or implicit security policy.*

While these definitions may be too relaxed for some purposes, they are adequate for our purposes here.

1.2 Objects and Roles

In general, a classification scheme takes the form that an **object** *has an* **attribute** that *has a* **value**. The bold words represent nouns and the italics represent a relationship between the nouns.

The nouns in our classification scheme can be artifacts such as a source code module, runtime library, or executable program image, or they can be more abstract concepts like vulnerabilities or security flaws. In any case, when defining attributes, it is important to be clear about what object the attribute is describing; otherwise it is easy to confuse a description of an exploit with a description of a vulnerability, for example.

In addition to showing the various types of objects for which properties can be attributed, Figure 1 illustrates the various actors and their associated roles. For example, a security analyst might be primarily concerned with different properties of security flaws and how to identify them. A programmer might be mainly concerned with the properties of the source code they are developing or maintaining, what it does, and whether it contains security flaws. A vulnerability analyst may primarily be concerned with analyzing vulnerabilities in existing and deployed programs. For a vulnerability classification scheme to be widely adopted, it has to be suitable by multiple users in multiple roles for multiple purposes.

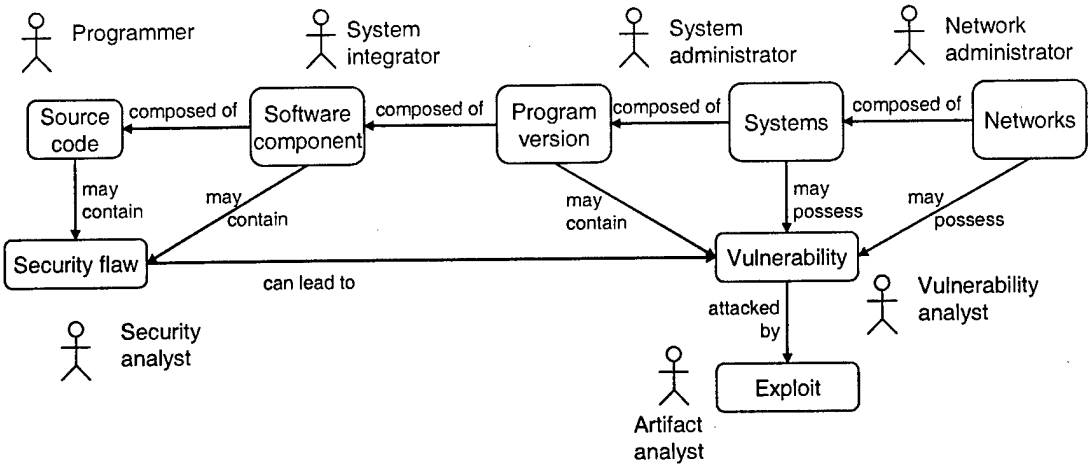


Figure 1: Objects, Roles, and Relationships

1.3 Existing Approaches

There are a number of existing approaches for classifying vulnerabilities. Many of these approaches are *taxonomies*.

A taxonomy is a system of classification that allows vulnerabilities to be uniquely identified. The best know example is the science of systematics, which classifies animals and plants into related groups.

Two major studies from the 1970s attempted to create taxonomies of security flaws. The RISOS study [Abbott 76] focused on flaws in operating systems; the other, the Program Analysis (PA) study [Bisbey 78], included both operating systems and programs. Interestingly, the taxonomies both presented were similar, in that the classes of flaws could be mapped to one another. Since then, other studies have based their taxonomies on these results [Bishop 95, Landwehr 94] (Table 1 illustrates another example of software vulnerabilities classification from Landwehr, Bull, McDermott, and Choi). However, the classifications defined in these studies are not "real" taxonomies in the sense that they fail to define classification schemes that identify a unique category for each vulnerability.

Aslam's recent study [Aslam 95] approached classification slightly differently, through software fault analysis (a decision procedure determines into which class a software fault is placed). Even so, it suffers from flaws similar to those of the PA and RISOS studies.

Table 1: Types of Software Vulnerabilities

Intentional	Malicious	Trojan Horse	Non-replicating
			Replicating (virus)
		Trapdoor	
		Logic/Time Bomb	
	Non-malicious	Covert Channel	Storage
			Timing
		Other	
Inadvertent	Validation Error (Incomplete/Inconsistent)		
	Domain Error (Including Object Reuse, Residuals, ...)		
	Serialization/aliasing (including race conditions, TOCTTOU errors)		
	Identification/Authentication Inadequate		
	Boundary Condition Violation (including resource exhaustion, ...)		
	Other Exploitable Logic Error		

Figure 2 illustrates a subset of the existing CERT® [1] Vulnerability Taxonomy. Unfortunately, this scheme also has significant flaws that make it unsuitable for continued use. Vulnerabilities are included in multiple categories, making it impossible to determine frequency data. The classification scheme is based on vulnerability reports and not on an engineering analysis of the problem domain. There is no clear correlation between these categories and avoidance strategies that can be used to prevent or limit vulnerabilities. Although approximately 800 vulnerabilities have been classified according to this scheme, the implementation is poorly designed and is now largely abandoned.

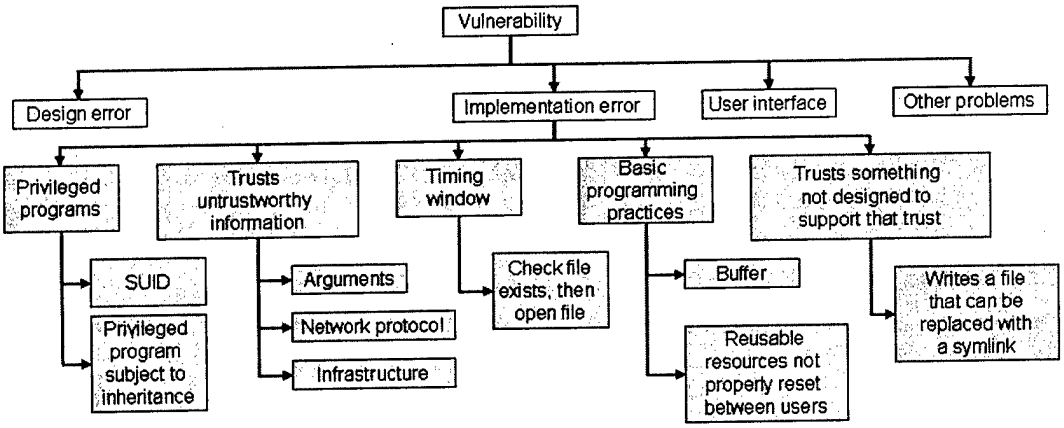


Figure 2: CERT Vulnerability Taxonomy (subset)

1. CERT is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

1.4 Properties

The classification scheme described in this report uses attribute-value pairs instead of a hierarchical taxonomy. Once an object has an attribute with a defined value it becomes a *property* of that object. Attribute-value pairs eliminate the problem of having vulnerabilities that fit into multiple classifications and thus invalidate frequency data. Instead, multiple attribute-value pairs can be associated with a software component to provide an overall picture of the security of that component.

A software program, of course, has many attributes. These attributes include size, complexity, performance, reliability, robustness and other quality attributes. For our purposes we are only interested in attributes that characterize the overall *vulnerability* of a program. These attributes can represent security flaws (that may or may not lead to vulnerabilities) as well as qualities of the overall component that lead to enhanced security. Of particular interest are attributes that are associated with known *exploits* and known *mitigation techniques.*

Exploits and Vulnerabilities

Figure 3 contains a sample program with some associated attribute value pairs. This is not a real program but a simple example of some code with obvious vulnerabilities. In particular, the program contains multiple security flaws, including an unbounded string copy and insufficient input validation on input arguments. Another attribute of this program is that it uses a memory manager that uses boundary tags, such as Doug Lea's malloc or Microsoft's RtlHeap. These two security flaws, combined with the use of a memory manager that uses boundary tags, leads to a *potential vulnerability.*

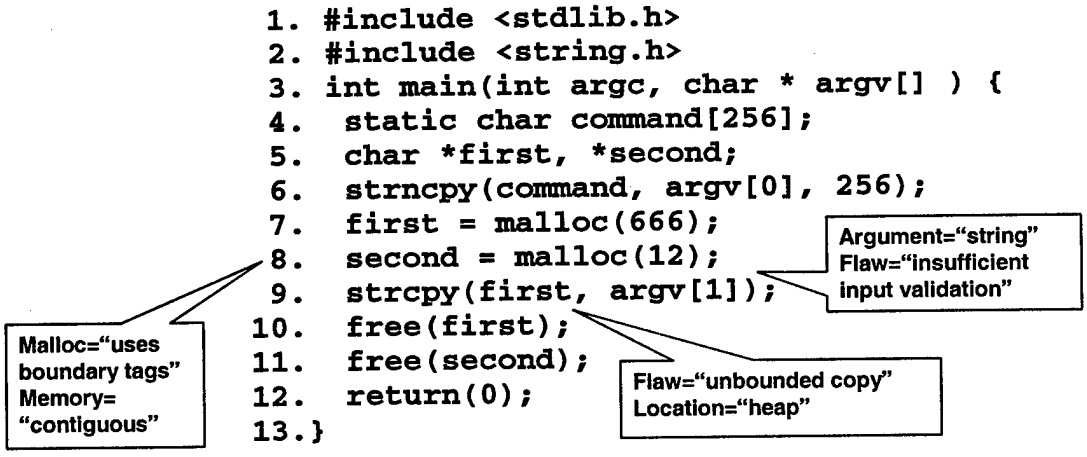


Figure 3: Sample Vulnerability Attributes

Exploits and vulnerabilities are, of course, inherently entwined. For example, the sample program in Figure 3 can be exploited by passing a malicious argument greater than the size of the first memory chunk. In this case, the unbounded string copy on line [9] will overwrite the end of the buffer and boundary tags at the end of the first chunk and at the front of the second chunk, as illustrated by Figure 4. These boundary tags can be overwritten in a manner that will cause an arbitrary address to be overwritten by another arbitrary address on the next call to `free()`. As a result, an attacker can exploit this vulnerability to transfer control to arbitrary code that may be part of this string or inserted elsewhere in memory. This particular exploit is diagramed in Figure 5 as a Unified Modeling Language (UML) activity diagram.

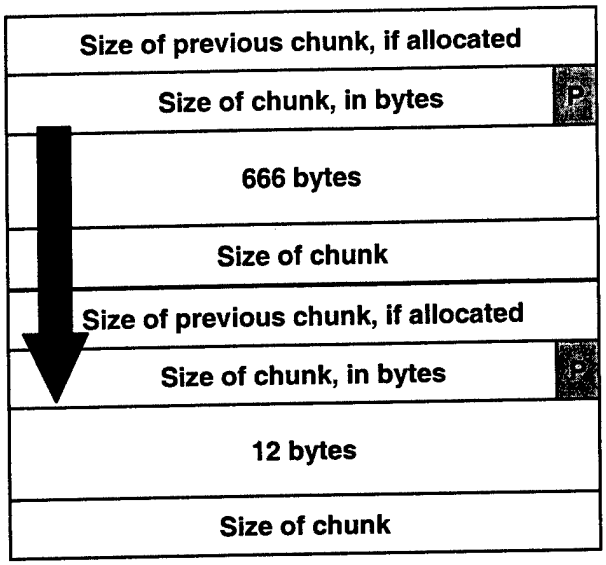


Figure 4: Overwriting Boundary Tags

The activity diagram for the exploit illustrates the relationship between the exploit and the vulnerable program. In particular, each activity is tagged with an attribute-value pair that represents the necessary preconditions for the exploit to succeed. Not surprisingly, these preconditions all exist in the vulnerable program from Figure 3.

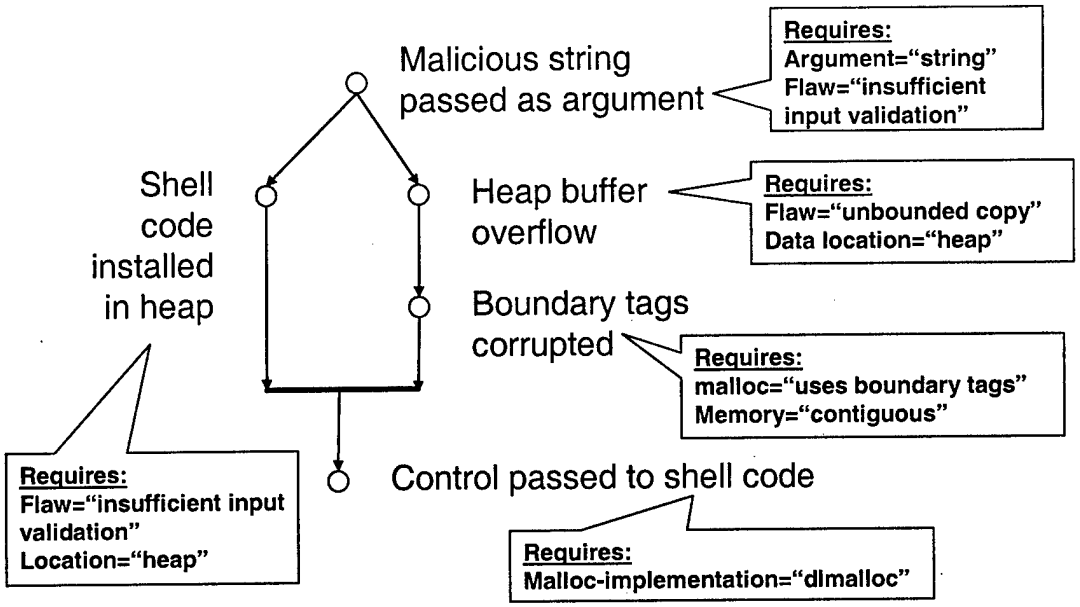


Figure 5: UML Activity Diagram of Exploit

The relationship between vulnerable programs and exploits has important consequences. First, it makes sense to develop a common set of attributes (and valid values) that can simultaneously be used to describe both vulnerabilities and exploits. Second, if a database of attributes existed for programs with known vulnerabilities or other security implications, it would be possible to automatically produce a list of programs that might be vulnerable to a known exploit by matching the attributes required by the exploit with the known attributes for a program. Theoretically, this could also be used to evaluate existing programs against new (previously unknown) vulnerabilities, presuming the relevant attributes required by this exploit have been recorded.

Mitigations and Vulnerabilities

Relationships also exist between vulnerabilities and *mitigations* (and between exploits and mitigation techniques as well). Mitigations[2] include methods, techniques, processes, tools, and runtime libraries that can prevent or limit exploits against vulnerabilities. Mitigation may work by

- eliminating a property of a program that represents a security flaw or other precondition necessary to create a vulnerability
- preventing an exploit from achieving a required property

2. Alternatively referred to as countermeasures or avoidance strategies.

As a result, mitigations can be implemented on different objects at different levels of abstraction. For example, a mitigation may be applied at the source code level that eliminates a security flaw and the associated vulnerability, or a work-around can be applied at a system or network level to prevent the security flaw from being exploited (also eliminating the vulnerability).

There are many examples of each type of mitigation technique. For example, it is possible on some platforms to install operating system patches to create non-executable stack segments or non-executable heap segments. This mitigation technique does not modify any of the properties of the vulnerable program illustrated in Figure 4, but it does affect the exploit diagramed in Figure 5. By preventing the heap segment from containing an executable, this particular mitigation technique prevents execution of the shell code that is installed in the heap. It is important to note in this case that preventing this one exploit by no means makes this code "secure," as there are other variants of this exploit that could still succeed.

An example of a mitigation that works by modifying the properties of the vulnerable program would be replacing calls to `strcpy()` with calls to `strncpy()`. In the case of the vulnerable program illustrated in Figure 4, this would eliminate the "unbounded copy" flaw and also eliminate the vulnerability.

The relationships between the properties displayed by software programs, those required by exploits, and those eliminated or prevented by mitigations increases the value of determining and recording these properties. By defining properties of vulnerable code, exploits, and mitigation techniques we can better determine which mitigation strategies will secure a vulnerable program against which exploits.

1.5 Classification Issues

Issues that further complicate classification include *levels of abstraction* and *viewpoint* [Bishop 96]. The xterm logging vulnerability described in CERT Advisory CA-1993-17 can be used to illustrate both issues.

The xterm program emulates a terminal under the X11 window system running as root on most UNIX systems. It enables the user to log all input and output to a file. If the file does not exist, xterm creates the log file and makes it owned by the user. If the file already exists, xterm checks that the user can write to it before opening the file. As any root process can write to any file on the system, the extra check is necessary to prevent a user from having xterm append log output to (say) the system password file and gain privileges by altering that file.

The following code fragment opens the file for writing when a user logs I/O to an existing file:

```
if (access("/usr/rcs/out", W_OK) == 0){
  fd = open("/usr/rcs/out", O_WRONLY|O_APPEND);
```

The semantics of the UNIX operating system cause the name of the file to be loosely bound to the data object it represents, and the binding is asserted each time the name is used. If the data object corresponding to `/usr/rcs/out` changes after the access but before the open, the open does not open the file checked by access. During that interval an attacker deletes the file and links a system file (such as the password file) to the name of the deleted file. Then xterm appends logging output to the password file. At this point, the user can create a root account without a password and gain root privileges.

At the lowest level of abstraction this vulnerability could be classified as an input validation problem, since the programmer fails to ensure that the object being validated is the same object the (potentially) insecure operation is performed on. At the next higher level of abstraction, this vulnerability could be viewed as a *race condition* vulnerability. At an even higher level of abstraction, this vulnerability could be classified as a logic or design error, since a resource (in this case, the file) can be deleted while in use.

Viewpoint is also important when classifying a vulnerability. From the perspective of an operating system developer, the vulnerability may be classified as a lack of required atomicity in the operations. Since changing the OS is not an option for an application developer, this problem may be classified as a problem with unnecessarily elevated privileges.

1.6 Credentials

Since experience has proven that taxonomies are not particularly conducive to classifying vulnerabilities, the classification scheme proposed in this report is based on attribute-value pairs instead. Attributes and values are selected based on engineering distinctions that allow vulnerabilities to be exploitable by a given exploitation technique or prevent them from being exploitable and that allow countermeasures to work or prevent them from working. In addition, we borrow the idea of credentials from Mary Shaw [Shaw 96] to indicate the confidence we have in the correctness of each attribute-value pair. In this way, credentials allow us to differentiate between knowledge and information.

> **Definition: Credential.**
>
> A credential is a triple <property, value, confidence>, where *property* is the name of the security property, *value* is the value of this property for a particular application, and *confidence* is a measure of the confidence we have in this information.

An example of a credential might be <security-flaw, unbounded-string-copy, third-party-report>. This credential states that a software component contains a security flaw that can be classified as an unbounded string copy and that this information has been reported by a third party, which implies a level of confidence. For example, experimentally validating a finding in a lab might promote the highest degree of confidence, followed by a vendor confirming an issue, and so on down to "someone I know thought they read that on a news group." Credentials are meant to expose the distinction between knowledge and information, or what a component *really does* versus what we *think* it does. How knowledge about security properties is obtained affects the confidence we have in that knowledge.

1.7 Attribute Sets

When a classification scheme is designed to support multiple goals, it can become cumbersome for an individual user who intends to use it for a single purpose. The ability to define sets of related properties that are appropriate for different user roles can make a vulnerability classification less cumbersome for use by a particular individual to achieve a particular goal. For example, it may make sense for a development organization to differentiate between attributes that are related to *runtime*, *linkage*, and *source code*.

The runtime category refers to properties of a program that exist once the program has been deployed into a particular environment. The same program image can have different properties when installed in different environments (for example, as a result of dynamic runtime linkage). The runtime category is primarily of interest to system administrators who need to evaluate a product for *actual vulnerabilities* and not *potential vulnerabilities* or *security flaws*.

The linkage category refers to the properties of a program that exist once the program has been linked into an executable image but not yet deployed. Properties in this category are never actual vulnerabilities, since the software is not deployed. This category of properties includes potential vulnerabilities and security flaws resulting from source code being compiled and (statically) linked to existing libraries. A good example of linkage properties can be seen in the example from Section 1.4. In that example, the program is only vulnerable when the source code is linked with a vulnerable memory manager. If the vulnerable library or component is statically linked, this property belongs in the linkage category. If the vulnerable library or component is dynamically linked, this property belongs in the runtime category. Linkage properties are of interest to system integrators and system administrators (when they can result in an actual vulnerability).

The source code category refers to properties of the source code before compilation and linkage. Properties in this category are primarily software flaws. For example, a C++ class or method may contain a buffer overflow. Because the method containing the security flaw is not used or only called using static data, there is no possibility of an actual vulnerability occurring,

so the problem is assigned a lower priority and not resolved. However, if this same C++ class is reused in another program, or the existing program is modified to use this method in an insecure manner, the result could easily become a fielded vulnerability. Source code properties are of primary interest to software engineers and quality assurance personnel, but may also be of interest to system administrators when these properties lead to runtime vulnerabilities.

Other attribute sets can be defined for other user roles, including vulnerability analyst, triage, and vulnerability handler.

2 Security-Related Software Attributes

This section describes the properties of objects that are critical to understanding software security. These properties are based on an in-progress evaluation of C/C++ implementation-level vulnerabilities and hence are inherently incomplete. No attempt has been made in this section to classify vulnerabilities resulting from design errors, insecure configurations, incorrect uses of cryptography, and other leading causes of vulnerabilities.

2.1 Source Code

Security flaws that can lead to software vulnerabilities exist in, and are normally repaired in, source code. Source code is primarily the responsibility of programmers, who are responsible for its development and maintenance.

Table 2 shows the *illicit control transfer mechanism* attribute of C and C++ source code. The values for this attribute are known mechanisms for an attacker to cause a program to execute arbitrary code. These mechanisms readily correlate to commonly used vulnerability classes (for example, buffer overflows are an example of writing beyond array bounds). Writing freed memory, freeing unallocated or non-heap memory, and user-supplied format strings are all higher level mechanisms for accomplishing a single-purpose: writing an arbitrary value to an arbitrary address. The values listed in Table 2 are easier to detect and label in C and C++ source code. This is the first example in which we have made a decision regarding the appropriate level of abstraction to expose in our classification scheme.

Table 2: Illicit Control Transfer Attribute

Attribute	Values
Illicit control transfer mechanism	Writing beyond array bounds, writing freed memory, freeing unallocated or non-heap memory, user-supplied format string

We use this as a starting point for our analysis, since gaining access to one of these illicit control transfer points is the enabling mechanism for most exploits.

Memory Properties

All the illicit control transfer mechanisms listed in Table 2 involve overwriting memory. The principle differences are where this memory is located and how it is overwritten. Although the exact layout of process memory is operating system specific, generally speaking memory exists in either the *stack*, *heap*, or *data* segments. A successful exploit must overwrite memory for a purpose. The purpose may be to modify the value of a *variable*, *data pointer*, *function pointer*, or *return address* on the stack. Modification of a a variable may be used to change some behavior of a program, possibly making it vulnerable to further attack. Modification of a data pointer, function pointer, or return address can all be used to execute arbitrary code. These attributes and valid values for these attributes are shown in Table 3.

Table 3: Compromised Memory Location and Description

Attribute	Values
Overwritten memory location	stack segment, heap segment, data segment
Data type modified	variable, data pointer, function pointer, return address

Functional Interface

In many cases, software vulnerabilities result from the incorrect use of a particular function or class of functions. This property is of interest because it can often be corrected by using a different but related function or by using an entirely different data abstraction that provides a similar capability. Table 4 defines properties for insecurely used functions. The attributes are common flaws, while the values are the actual function names.

Table 4: Functional Interface Properties

Attribute	Values
Unbounded memory copy	`strcpy()`, `memcpy()`, `sprintf()`, etc.
Incorrect length	`strncpy()`, `snprintf()`, etc.
User-provided sensitive argument	`printf()`, `setuid()`, etc.

A mapping to the functional category (e.g., string manipulation, dynamic memory management, formatted I/O, file I/O) can be made based on the value of the insecurely used function.

Integer Operations

Fairly recently, a number of vulnerabilities have been attributed to exceptional conditions related to integer operations or a failure to adequately constrain the range of an integer value. These security flaws cannot be directly exploited but generally allow an attacker to create or access one of the illicit transfer control mechanisms listed in Table 2, such as a buffer overflow. The effected integer is often used as *sizes*, *array indices*, or *loop counters*. Integers can also be *multipurpose* or used in *other* ways.

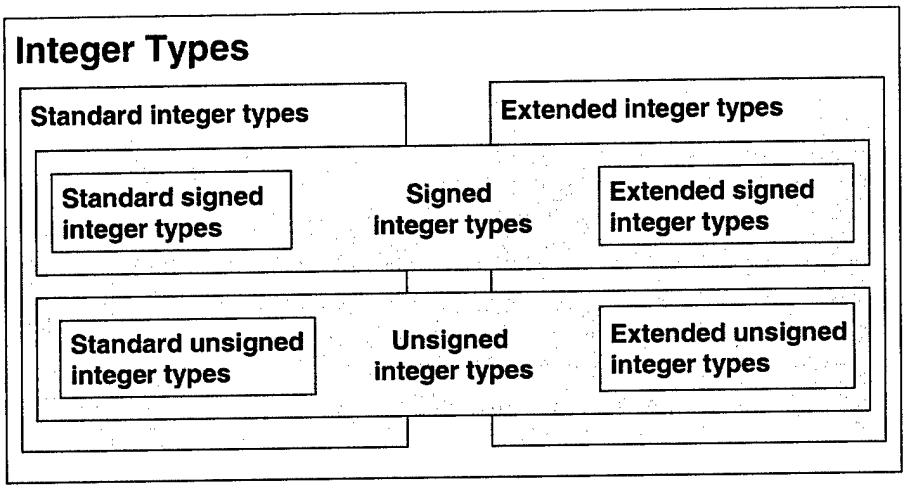


Figure 6: Integer Types

It is also useful to determine whether the integer vulnerability is the result of an exception condition or is simply the result of insufficient range checking. The possible error conditions are *overflow*, *sign*, *truncation*, and *insufficient range checking*. (If the vulnerability is the result of faulty logic, a "no error" condition exists and the error condition is specified as *none*.) Another important characteristic about an integer type range exploit is the type of the integer being attacked. Integers are organized into sets, as illustrated in Figure 6. The size and signedness of the vulnerable integer are also classified.

Table 5: Integer Range Error Classification Scheme

Property	Values
Integer application	array index, loop counter, size, multipurpose, other
Integer error	overflow, sign, truncation, insufficient range checking
Integer set	standard, extended
Integer signedness	signed, unsigned

Table 5: Integer Range Error Classification Scheme

Property	Values
Standard integer type	`char`, `short int`, `int`, `long int`, and `long long int`
Extended integer type	int8_t, uint24_t, int_least32_t, uint_least16_t, int_fast16_t, uint_fast64_t, intptr_t, uintptr_t, intmax_t, uintmax_t, etc.

Table 5 lists the classification properties and possible values. The integer range error classification can be used to evaluate whether a vulnerability can be prevented by a particular avoidance strategy.

Format String Vulnerabilities

Format string vulnerabilities can be exploited to run arbitrary code on a machine without overflowing a buffer, so there are clearly cases where these vulnerabilities are a uniquely separate class.

Format string exploits work by taking either *partial* or *complete* control of the format string. Exploits that do not control the format string are normally buffer overflows. Formatted input/output exploits should also be differentiated between *input* and *output* functions because each group of functions shares a different specification and (typically) different implementations. Attributes and associated values related to format string vulnerabilities are listed in Table 6.

Table 6: Formatted Input/Output Classification Scheme

Attribute	Values
Input/output function	input, output
Format string control	none, partial, complete

2.2 Software Components

Software components are the elements from which larger software programs are composed [Wallnau 01]. Software components include shared libraries such as Dynamic Link Libraries (DLLs), ActiveX controls, Enterprise JavaBeans, and other compositional units. Software components may be linked into a program or dynamically bound at runtime. Software components, however, are not directly executed by an end user, except as part of a larger program. Therefore software components cannot have vulnerabilities because they are not executable outside of the context of a program.

2.3 Program Versions

Program versions are actual executable images that can be installed and run on a system. Products, on the other hand, are a marketing abstraction that encompasses both past and future versions of a program. Therefore, program versions can have vulnerabilities, while products cannot.

2.4 Mitigations

A mitigation is a solution for a software flaw or vulnerability or a work-around that can be applied to prevent exploitation of a vulnerability. At the source code level, mitigations may be as simple as replacing an unbounded string copy operation with a bounded one. At a system or network level, a mitigation might involve turning off a port or filtering traffic to prevent an attacker from accessing a vulnerability.

Identifying the attributes of mitigations is an important step in determining which vulnerabilities can be resolved by which mitigations.

2.5 Security Flaws

Security flaws are defects in source code or software components that can lead to software vulnerabilities. Security flaws have their own properties that describe the possible consequence of these flaws if exposed as vulnerabilities in a program. An understanding and analysis of security flaws is important to determine which programs may contain related vulnerabilities and what the possible consequence of these vulnerabilities may be.

2.6 Vulnerability Properties

Programs, systems, and networks exhibit vulnerabilities. Vulnerabilities are of interest to vulnerability analysts, vulnerability handlers, and system and network administrators. The underlying vulnerabilities that cause the vulnerabilities are of interest to software developers. As a result, the interesting properties of vulnerabilities are significantly different than the interesting properties of software flaws. Table 7 enumerates these attributes and associated values.

2.7 Exploit Properties

For the exploit to execute code, the code must already *exist* in the address space of the vulnerable process (presumably in the code segment) or it must be *injected.* Code could be injected into the *stack*, *heap*, or *data* segments. Where the code is injected can be relevant if one or more memory segments is made to be non-executable. Exploits can also be differentiated

Table 7: Vulnerability Properties

Attribute	Values
Impact	mislead application users, denial of service, crash system, destroy data, read protected information, create files used by others, gain access to many users, obtain super user/administrative access
Affected product	status unknown, vulnerable, not vulnerable
Solution	Upgrade, apply patch, use an alternative product
Extent known	restricted, solutions released, general concept public, public
Required to exploit	access to privileged account, trusted host, nearby host, local access to user account, any remote user using an uncommon service, any remote user using a common service (e.g., Web, FTP)

based on their consequence. Exploits can be used to *crash* a program, *read memory*, *write memory*, or *execute arbitrary code*.

These exploit attributes and valid values for these attributes are shown in Table 8.

Table 8: Exploit Properties

Attribute	Values
Exploit code	injected, existing
Exploit code location	stack segment, heap segment, data segment
Consequence	crash program, read memory, write memory, execute arbitrary code

3 Representation and Automation

This section describes possible representations for the classification of properties related to vulnerabilities, software flaws, exploits, and other security-related software objects.

3.1 Representing Properties

The Resource Description Framework (RDF)[3] is a framework for representing information in the Web. RDF [Klyne 04] was developed for use in

- Web metadata: providing information about Web resources and the systems that use them (e.g., content rating, capability descriptions, privacy preferences, etc.)
- applications that require open rather than constrained information models (e.g., scheduling activities, describing organizational processes, annotation of Web resources)
- doing for machine processable information (application data) what the Web has done for hypertext: allow data to be processed outside the particular environment in which it was created, in a fashion that can work at Internet scale
- interworking among applications: combining data from several applications to arrive at new information
- automated processing of Web information by software agents: The Web is moving from having just human-readable information to being a world-wide network of cooperating processes. RDF provides a world-wide *lingua franca* for these processes.

RDF is designed to represent information in a minimally constraining, flexible way. It can be used in isolated applications, where individually designed formats might be more direct and easily understood, but RDF's generality offers greater value from sharing. The value of information thus increases as it becomes accessible to more applications across the entire Internet.

3. See http://www.w3.org/RDF/.

In short, RDF allows us to represent statements of the form "subject predicate object," where "predicate" indicates the relationship between subject and object. For example, an individual initially triaging a vulnerability report might add the following data:

- **John Smith** *reported* **VU#999999**
- **VU#999999** *describes* a **vulnerability**
- **VU#999999** *describes* a **buffer overflow**
- **VU#999999** *affects* the **foo html library**

while because of prior work, there might already be a body of knowledge captured by this:

- The **foo html library** *is a component of* the **IE rendering engine**
- The **IE rendering engine** *is an implementation of* an **HTML rendering engine**
- The **IE rendering engine** *is a component of* **Internet Explorer**
- **Internet Explorer** *is an implementation of* a **Web browser**
- **Internet Explorer** *is a product of* **Microsoft**
- **Firefox** *is an implementation of* a **Web browser**
- **Firefox** *is a product of* the **Mozilla Foundation**
- **HTML rendering engines** *are a component of* **Web browsers**

Thus, given the above, an automated system could infer that Microsoft should be alerted to this vulnerability and that further investigation may be warranted by the Mozilla Foundation to ascertain whether Firefox is affected by VU#999999.

On further analysis, perhaps the analyst discovers

- **VU#999999** *is* **remotely exploitable**

which, coupled with the previous existing knowledge, as well as the following:

- **remote exploitation** *implies* **virus potential**
- **Internet Explorer** *is a* **widespread product**
- **vulnerabilities in widespread products** *are at* **high risk of exploitation**

allows an automated system to infer that VU#999999 has significant risk of appearing in a future virus, as shown in Figure 7.

The Web Ontology Language (OWL) [McGuinness 04, Patel-Schneider 04] extends RDF by providing a standardized vocabulary with which one can describe a number of semantically meaningful relationships (for example, "*x* hasParent *y*." "*x* hasSibling *z*."—which allows for automated reasoning to deduce that "*z* hasParent *y*").

Although much work would need to be done to realize a finished product, we believe that the classification scheme described in this paper could be represented using RDF and OWL meta-

data such that emerging tools in that space could be used to navigate and mine vulnerability data.

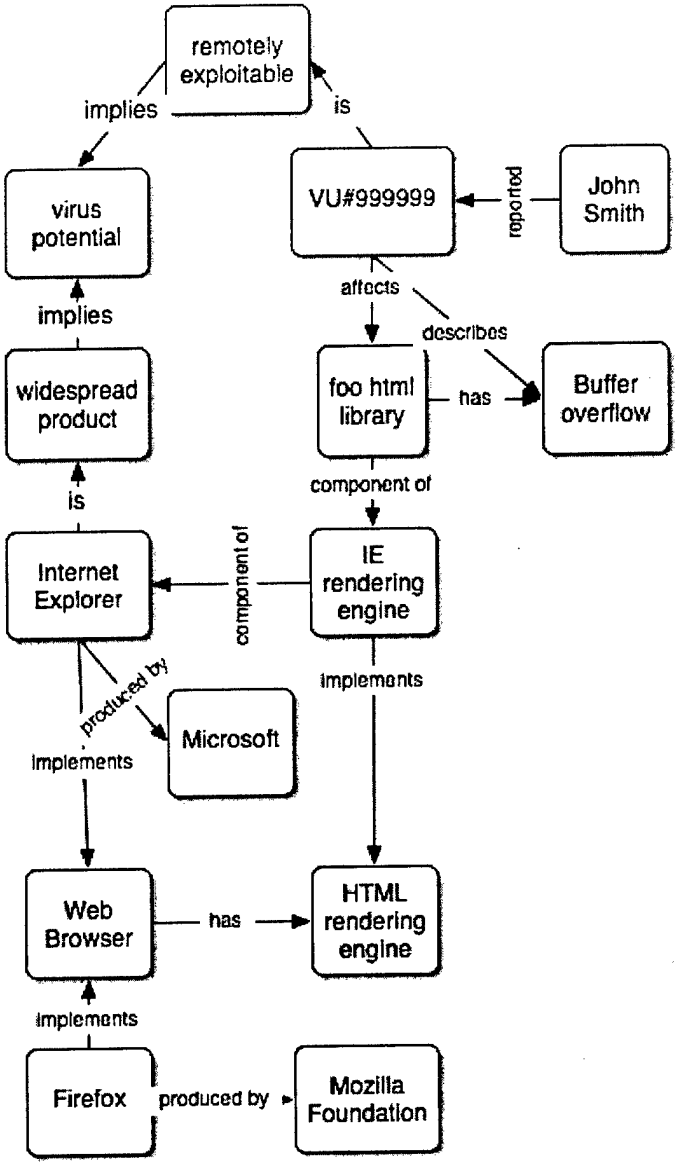


Figure 7: Complete RDF Example

3.2 Comparative Analysis of Vulnerabilities

System and network administrators, vendors, service providers, researchers, and computer security incident response teams (CSIRTs) regularly receive new vulnerability reports whose

severity needs to be assessed. The information available is almost always, by necessity, incomplete. Regardless, there is an urgent need to prioritize handling of the vulnerability. Could a database of existing vulnerabilities with known properties be used to assess the new report?

Applying existing vulnerability knowledge requires that we can compare the new vulnerability to the set of already known vulnerabilities. First, let's consider the general problem of describing the similarity of two vulnerabilities. Assuming that all attributes are binary (i.e., the vulnerability has or does not have a particular attribute), we can represent the set of attributes for a vulnerability as a bitmap. Doing so allows us to compare two vulnerabilities based on their attribute sets by XORing their bitmaps. This is equivalent to calculating the Hamming Distance[4] between the two bitmaps.

For example, given a classification system with fivebinary attributes, two vulnerabilities, Vul1 and Vul2, can be compared as follows:

```
01110 = Vul1's bitmap
10110 = Vul2's bitmap
-----
11000 = XOR(Vul1,Vul2)
```

Vul1 and Vul2 differ in two of five attributes, so the hamming distance is calculated as (2/5) = 0.4. To access a vulnerability report against a database of N vulnerabilities, it may be necessary to to calculate N^2 distances. However, it is conceivable that the problem could be reduced if it were possible to identify *landmark* vulnerabilities that could be used as reference points. This would allow for a new vulnerability to be compared to some subset of all vulnerabilities to assess its relative position in the vulnerability space. The number of landmarks that would be required is one more than the dimensionality of the data, but since we don't have any actual data, we cannot make any statements regarding the number of dimensions required to represent it. (The individual attributes in a set are unlikely to be completely orthogonal, thus it's not necessarily true that 10 attributes implies 10 dimensions.)

Now, a new vulnerability is reported, which we'll call Vul3. We don't know much about Vul3 yet, but we can at least specify three of its attributes, so we represent its bitmap as

```
101?? = Vul3's bitmap
```

4. In information theory, the Hamming distance is the number of positions in two strings of equal length for which the corresponding elements are different. Put another way, it measures the number of substitutions required to change one into the other. It was named after Richard Hamming. The Hamming distance is used in telecommunication to count the number of flipped bits in a fixed-length binary word, an estimate of error, and so is sometimes called the signal distance. It corresponds to the weight (number of ones) in the XOR of the words, or to the Manhattan distance between two vertices in an n-dimensional hypercube, where n is the length of the words.

Where "?" indicates the "don't know" bits.

Even given incomplete information about Vul3, we can still estimate its distance from Vul1 and Vul2 by calculating the possible range of values:

```
01110 = Vul1's bitmap        10110 = Vul2's bitmap
101?? = Vul3's bitmap        101?? = Vul3's bitmap
-----                        -----
110?? = XOR(Vul1,Vul3)       000?? = XOR(Vul2,Vul3)
```

The distance between Vul1 and Vul3 is at least 0.4 (2/5 of the bits differ), and its maximum distance is 0.8 (if both unknown bits turn out to be different, too). Similarly, Vul2 and Vul3 have a minimum distance of 0 and a maximum distance of 0.4. It is apparent even before the other attributes are known that Vul3 has more in common with Vul2 than with Vul1.

Earlier we assumed that all attributes are binary, in which case a simple XOR operation across a bitmap would suffice to determine the distance between two vulnerabilities. The more general case where attributes may take on a range of values can be addressed in one of two ways:

1. Multivalued attributes can be translated into a series of binary attributes, in which case the operation described above would still apply, or
2. A function can be defined for each non-binary attribute to calculate the *match*. In the binary version, attributes either matched (a 0 bit in the XOR result), or didn't (a 1 bit in the XOR result). But recall that the distance metric was based simply on summing up the number of 1s in the XOR result. If instead we described a match as having a value between 0 and 1, then we could represent a partial match for a non-binary attribute. To calculate the distance, one would sum up the partial matches, and the distance calculation is still given by the sum of the matches (partial or complete) divided by the number of attributes.

Given sufficient vulnerabilities with known qualities, it is possible to correlate attributes. For example, it is possible to calculate the probability that a vulnerability will have high severity given that a particular attribute is, or is not, set (the same sort of analysis could be done for sets of attributes). Given our previous example, knowing the severity of a vulnerability similar to Vul3 allows a researcher to approximate the severity of Vul3. It may also be possible to infer the likelihood that a given vulnerability will be exploited by a particular type of artifact (e.g., bot, worm, virus).

4 Conclusions

Vulnerability classification must be based on solid engineering analysis to be useful in determining the threat represented by the vulnerability and, consequently, predicting any future threat.

Classification can enable real-world benefits, including automatic assessment of threat posed by vulnerabilities and assessment of mitigation strategies and techniques. Formalizing classification and analysis of vulnerabilities should make it easier to share information among geographically distributed organizations. Vulnerability analysts create formal descriptions of known exploits. Vulnerability remediation specialists can then create formal descriptions of suspect source code and analyze it using tool sets (reducing the level of experience required).

Today vulnerability analysis is ad hoc and depends on the skills and inclinations of vulnerability remediation specialists. Vulnerability analysis process should reduce dependency on knowledgeable analysts. Association of values with attributes becomes the goal of analysis. Unidentified traits result in an extension or reevaluation of the classification scheme.

Attributed code segments can be automatically analyzed against the classifications to determine whether they are vulnerable to any known class of exploit. Code previously considered not vulnerable can be automatically reevaluated when new exploits are discovered. Mitigations can be evaluated to determine which exploits are prevented and which exploits are *not* prevented.

Vulnerability classification will allow increased standardization of vulnerability analysis, which in turn will allow for greater sharing of information and opportunities for automation.

References

[Abbott 76] Abbott, R. P.; Chin, J. S.; Donnelley, J. E.; Konigsford, W. L.; Tokubo, S.; & Webb, D. A. "Security Analysis and Enhancements of Computer Operating Systems." NBSIR 76-1041, Institute for Computer Sciences and Technology, National Bureau of Standards, April 1976.

[Aslam 95] Aslam, T. "A Taxonomy of Security Faults in the UNIX Operating System." Master of Science thesis, Department of Computer Sciences, Purdue University, 1995.

[Bisbey 78] Bisbey, R. II & Hollingsworth, D. *Protection Analysis Project Final Report* (ISI/RR-78-13, DTIC AD A056816). Marina del Rey, CA: University of Southern California Information Sciences Institute, 1978.

[Bishop 95] Bishop, M. *A Taxonomy of UNIX System and Network Vulnerabilities*" (Technical Report 95-10). Davis, CA: Department of Computer Science, University of California at Davis, 1995.

[Bishop 96] Bishop, Matt & Bailey, David. *A Critical Analysis of Vulnerability Taxonomies* (CSE-96-11). http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/ucd-ecs-96-11.pdf (September 1996).

[CCPSO 99] Common Criteria Project Sponsoring Organizations. *Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and General Model.* CCIMB-99-031 Version 2.1, August 1999. http://niap.nist.gov/cc-scheme/cc_docs/cc_v21_part1.pdf.

[Fithen 04] Fithen, William L.; Hernan, Shawn V.; O'Rourke, Paul F.; Shinberg, David A. "Formal Modeling of Vulnerability." *Bell Labs Technical Journal 8*, 4 (February 5, 2004): 173-186.

[Klyne 04] Klyne, Graham & Carroll, Jeremy, eds. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. http://www.w3.org/TR/rdf-concepts/ (February 2004).

[Landwehr 94] Landwehr, C. E.; Bull, A. R.; McDermott, J. P.; & Choi, W. S. "A Taxonomy of Computer Program Security Flaws." *Computing Surveys 26*, 3 (September 1994): 211-255.

[McGuinness 04] McGuinness, Deborah L. & van Harmelen, Frank. *OWL Web Ontology Language Overview*. http://www.w3.org/TR/owl-features/ (February 2004).

[Patel-Schneider 04] Patel-Schneider, Peter F.; Hayes, Patrick; & Horrocks, Ian. *OWL Web Ontology Language Semantics and Abstract Syntax*. http://www.w3.org/TR/owl-semantics/ (February 2004).

[Shaw 96] Shaw, Mary. "Truth vs. Knowledge: The Difference Between What a Component Does and What We Know it Does," 181-185. *Proceedings of the 8th International Workshop on Software Specification and Design*. Schloss Velen, Germany. March 22-26, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996.

[Wallnau 01] Wallnau, Kurt C.; Hissam, Scott A.; & Seacord, Robert C. *Building Systems from Commercial Components*. Boston, MA: Addison-Wesley, June 2001 (ISBN: 0201700646).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 2005	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
A Structured Approach to Classifying Security Vulnerabilities	F19628-00-C-0003
6. AUTHOR(S)	
Robert C. Seacord & Allen D. Householder	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2005-TN-003

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12.a DISTRIBUTION/AVAILABILITY STATEMENT	12.b DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (maximum 200 words)

Understanding vulnerabilities is critical to understanding the threats they represent. Vulnerabilities classification enables collection of frequency data; trend analysis of vulnerabilities; correlation with incidents, exploits, and artifacts; and evaluation of the effectiveness of countermeasures. Existing classification schemes are based on vulnerability reports and not on an engineering analysis of the problem domain. In this report a classification scheme that uses attribute-value pairs to provide a multidimensional view of vulnerabilities is proposed. Attributes and values are selected based on engineering distinctions that allow vulnerabilities to be exploited by a given technique or determine which countermeasures are effective. Successful classification of vulnerabilities should lead to greater automation in analyzing code vulnerabilities and supporting effective communication between geographically remote vulnerability handling teams and vendors.

14. SUBJECT TERMS	15. NUMBER OF PAGES
vulnerability classification, exploit classification, vulnerability properties, vulnerability attributes	38
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102